Journal of Universal Computer Science, vol. 20, no. 5 (2014), 772-796
submitted: 28/7/13, accepted: 15/2/14, appeared: 1/5/14 © *J.UCS*

Thesaurus-Based Tag Clouds for Test-Driven Code Search

Otavio A. L. Lemos, Adriano C. de Paula, Gustavo Konishi
(Federal University of São Paulo, São José dos Campos, SP, Brazil
{otavio.lemos, adriano.carvalho, konishi}@unifesp.br)

Sushil Bajracharya
(Black Duck Software, Inc., USA
sbajra@acm.org)

Joel Ossher, Cristina Lopes
(University of California, Irvine, CA, USA
{josshers, lopes}@ics.uci.edu)

Abstract: Test-driven code search (TDCS) is an approach to code search and reuse that uses test cases as inputs to form the search query. Together with the test cases that provide more semantics to the search task, keywords taken from class and method names are still required. Therefore, the effectiveness of the approach also relies on how good these keywords are, *i.e.*, how frequently they are chosen by developers to name the desired functions. To help users choose adequate words in their query test cases, visual aids can be used. In this paper we propose *thesaurus-based tag clouds* to show developers terms that are more frequently used in the code repository to improve their search. Terms are generated by looking up words similar to the initial keywords on a thesaurus. Tag clouds are then formed based on the frequency in which these terms appear in the code base. Our approach was implemented with an English thesaurus as an extension to CodeGenie, a Java- and Eclipse-based TDCS tool. Our evaluation shows evidence that the approach can help improve the number of returned results, recall (by ∼28%, on average), and precision (by ∼14%, on average). We also noticed the visual aid can be especially useful for non-native speakers of the language in which the code repository is written. These users are frequently unaware of the most common terms used to name specific functionality in the code, in the given language.
Key Words: Test-driven code search, code search, software reuse, tag clouds
Category: D.2.13 - Reusable Software

1 Introduction

The increasing availability of open source code in the Internet has made possible code reuse through searches made upon open source software repositories [Bajracharya et al., 2010b]. Although this type of reuse can be relatively effective, it generally relies mostly on keywords, regular expressions, and other more syntactic information about the function to be found. Test-driven code search (TDCS) was proposed as a form of code search and reuse that makes use of more semantic information available on test cases[1] [Lemos et al., 2011].

[1] Similar test-driven code search approaches were also proposed by other researchers [Hummel et al., 2008; Reiss, 2009]

Although exploratory studies have shown that TDCS can be effective in the reuse of auxiliary functions, its success also relies on keywords extracted from test cases (*e.g.*, the searched method name). If the user selects these names poorly, few results will be returned. This issue is related to the *vocabulary mismatch problem* (VMP) as discussed by Bajracharya et al. [2010b]. VMP states that the likelihood of two people choosing the same term for a familiar concept is only between 10-15% [Furnas et al., 1987].

A way to circumvent this problem is to present different options to the user based on the initially chosen term. Similar keywords can be automatically investigated and presented visually to the user according to their frequency in the code base. Tag clouds are visual aids adequate for this context, because the size and color of the terms can be presented according to their relevance in the repository.

In this paper, we propose the use of thesaurus-based tag clouds to improve TDCS. A thesaurus of the same language in which the code in a given repository is written is used to explore similar terms for a given initial keyword taken from the method name. Each of these terms are searched in the repository and the tag cloud is formed according to their frequency in the code base. The terms are also weighted and colored according to where they appear in the full qualified names of matching methods. The closer the terms are to the method name, the larger their weight.

We implemented the proposed approach as an extension to CodeGenie [Lemos et al., 2011], a Java- and Eclipse-based TDCS tool. Since CodeGenie is based on Sourcerer [Linstead et al., 2009], an infrastructure with a code repository mostly in English, a thesaurus based on that language was used. To have an idea of the effectiveness of our approach, an exploratory study with example searches and a controlled experiment with 36 human subjects – including professional developers – were conducted. Our initial investigation shows that the tag clouds can improve the result set, recall (by ∼28%, on average), and precision (by ∼14%, on average), when the initially selected method name is replaced by a combination of the original and most frequent related terms. We also noticed that our approach can be specially useful for non-native speakers of the language in which the repository is based. These users may initially select terms that are not commonly used to name the desired functions, thus reducing the possibility to retrieve good results.

The remainder of this paper is structured as follows. Secion 2 presents backgound information about TDCS, CodeGenie, and Tag Clouds, and Section 3 presents our thesaurus-based tag clouds approach to TDCS. Section 4 presents details about our implementation using CodeGenie, and Section 5 presents an exploratory evaluation and a controlled experiment to evaluate our approach. Section 6 presents related work and, finally, Section 7 concludes the paper.

2 Background

The same way that test cases can be used to define a software feature in TDD, they can also be used to describe a desired feature in a code search task. Moreover, in this context, we can take advantage of the following characteristics of TDD [Erdogmus et al., 2005]:

1. Feedback: Test cases provide instant feedback about the suitability of a particular code result in the local context;
2. Task-orientation: The requirement of designing test cases first guides the developer in searching for self-contained and manageable software pieces, one at a time;
3. Quality assurance: Since code results might come from unknown sources which are not always trustable, tests cases help in assuring a certain degree of quality. Up-to-date test cases also helps keeping track of the quality of the retrieved software pieces along the evolution of the system.

TDCS makes use of test cases to describe a desired feature to be searched, the same way test cases are used in Test-Driven Development [Beck, 2002]. Figure 1 shows the basic TDCS process. To describe a missing feature in the project, test cases are designed in the Integrated Development Environment (IDE). The search facility can then be triggered and, based on the information available on the test cases, a query is sent to a code search service capable of processing it. In the IDE, the developer can explore the results by integrating/testing and detaching them. To do that, a program slicing service to provide self-contained code pieces and a repository access service must be available at the Code Services side. Whenever the developer feels satisfied with a particular code result, it can be left integrated to the project. Detaching of a code result at any time can also be done. [Lemos et al., 2011].

TDCS was implemented as an Eclipse[2] plugin named CodeGenie. The Code Services side is provided by Sourcerer, a source code infrastructure that provides all support needed to perform TDCS. CodeGenie formulates queries that contain three parts: (1) keywords present in the full qualified name of the desired method; (2) return type of the method; and (3) parameter types of the method. For example, given a test case with the assertion `assertEquals("trevni", Util.invert("invert"))`, CodeGenie formulates the following query:

fqn_contents:(util invert)
m_ret_type_contents:(String)
m_sig_args_sname:String

[2] `http://www.eclipse.org/` - accessed in 06-27-2012.

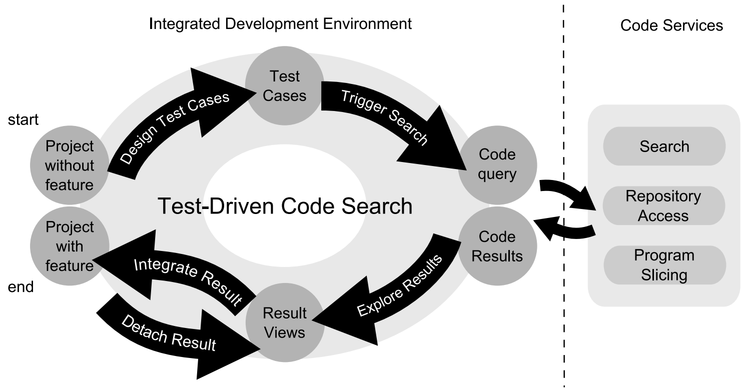

Figure 1: TDCS process [Lemos et al., 2011].

The query above means: "look for a method that contains the terms 'util' and 'invert' somewhere in the full qualified name, returns a String, and receives a String as a parameter".

2.1 Example

To show how TDCS is used in practice, we show an example of search conducted using CodeGenie [Lemos et al., 2011]. Consider the development of a document editing system. An important element of such systems are counters used to number sections, pages, etc. An Arabic to Roman function could be implemented to present counters as Roman numerals. Figure 2 presents sample test cases in JUnit for a static method implementation of the function. After implementing the JUnit test cases, the user triggers the CodeGenie search facility by right-clicking on the test class and selecting the CodeGenie Search menu option shown in Figure 3.

CodeGenie sends the query to Sourcerer which, in turn, returns code results. The keywords are initially formed by the method and class names. In the example, 'util' and 'roman' are the initial keywords. By default, every information on the test cases is used to generate the query, *i.e.*, class name, method name, and method signature. Figure 4 shows CodeGenie's Search View with the results related to the referred example.

The developer can examine, weave, and test a result by right-clicking on it and selecting 'Integrate Slice' and 'Test Slice'[3]. Integrated results can be detached by selecting the 'Detach Slice' option. When a result is integrated, it appears as '[currently integrated]' in the Search View, and it can be tested using

[3] When a result is integrated to the local workspace, the matched function name is renamed to conform to the terms used by the user in the test cases.

```
public class RomanTest {
  @Test
  public void testRoman1() {
    assertEquals("I", Util.roman(1));
  }

  @Test
  public void testRoman2() {
    assertEquals("II", Util.roman(2);
  }

  ...

  @Test
  public void testRoman6() {
    assertEquals("M", Util.roman(1000));
  }
}
```

Figure 2: Partial JUnit test class for an Arabic to Roman function.

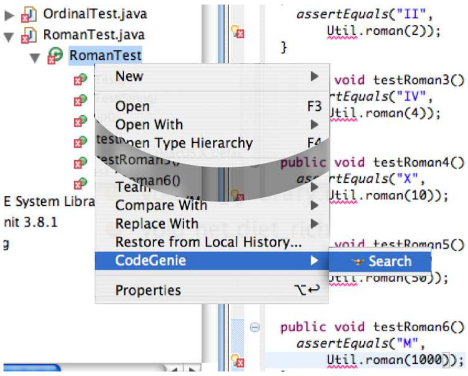

Figure 3: CodeGenie search being triggered [Lemos et al., 2011].

the test cases designed for the search. As results are tested, they are rearranged in the Search View, so that the most relevant appear first (in particular the ones that are successful against test cases). The user can also preview the code of a result by using the CodeGenie Snippet Viewer [Lemos et al., 2011].

Note that the query formed by CodeGenie also contains keywords that are important to its effectiveness. If the user ever selects them poorly, few results will be returned. To help obtaining good terms for keywords, visual aids such as tag clouds can be applied.

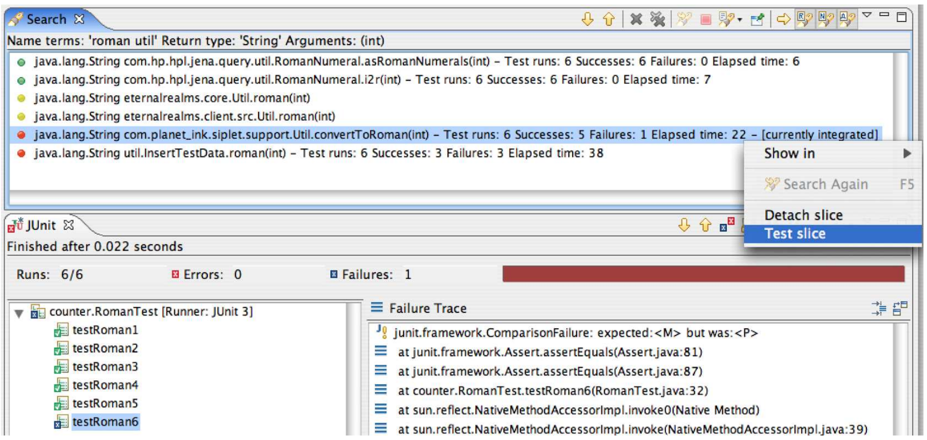

Figure 4: CodeGenie Search View [Lemos et al., 2011].

2.2 Tag Clouds

Tag clouds are visual presentations of a set of terms in which text attributes such as size, weight, or color are used to represent features of the associated terms [Rivadeneira et al., 2007]. Tag clouds can also be used to navigate in a set of terms, emphasize information, and to show results of a search task. This type of overview is helpful for unspecific retrieval tasks and also serves as a starting point for browsing, when users have no initial appropriate terms to start searching or browsing. According to Sinclair and Cardew-Hall [Sinclair and Cardew-Hall, 2008], using the tag cloud implies less cognitive and physical burden than thinking of search terms that define the thematic field to be explored, and entering them into a search field. Figure 5 presents an example of a tag cloud formed with terms related to the Web 2.0.

We believe this type of visual aid can be handy for TDCS, since terms have to be carefully chosen to name methods in the test cases, in order to obtain good recall.

3 Thesaurus-Based Tag Clouds for TDCS

Although initial evidence has shown that TDCS can be useful in the reuse of open source code [Lemos et al., 2011], some problems still affect its effectiveness. As discussed by Bajracharya et al. [2010b], in code search and reuse, the VMP mentioned earlier manifests itself as the gap between the situation and solution models [Fischer et al., 1991]. Developers implement code using words that describe the solution model while users seeking to reuse code might use words

Figure 5: Tag cloud sample adapted from [Cremonini, 2002].

that come from their situation models. In general purpose search engines such as Google, such mismatch is reduced by the abundance of linguistically rich documents. Information retrieval (IR) models work well on natural languages but tend to be ineffective with source code, which is linguistically sparse (*i.e.*, natural language documents have more complicated semantics compared to source code, and IR models are based on such type of documents; a simple example is when keywords used to look for specific function implementations match comments present in source code that has nothing to do with the desired behavior). Approaches such as TDCS take advantage of the richness of source code structure, but the VMP is still relevant because users also rely in the choice of good keywords.

A way to reduce this problem is to apply visual aids that can present similar terms to the ones initially chosen by users before the search task takes place. As commented earlier, tag clouds are specially suited for this context since they can present similar words with visual richness based on the frequency in which these words appear in the code repository. Moreover, as commented by Sinclair and Cardew-Hall [Sinclair and Cardew-Hall, 2008], users for whom English is a second language seem to find tag clouds particularly helpful when wanting to search for something without knowing where to start (which can frequently be the case when naming methods in the TDCS query test cases). In fact, the experiment presented in Section 5 shows evidence about such claim.

There are several ways to explore analogous keywords when constructing the tag clouds, such as using similarity algorithms like the one proposed by Bajracharya et al. [2010b]. In this paper we use a simpler approach that makes use of thesauri. The idea behind the thesaurus-based tag cloud is to form the cloud from synonyms of a given initial term, assigning weights to the returned results according to their relevance in the code base. In TDCS the tag cloud is used to search all synonyms of a given keyword taken from the method name. For example, while searching for a method initially named *sum* that adds two numbers, the cloud would show the words *addition*, *aggregation*, and *calculation*, according to the frequency in which each term appears in the repository. In this way, developers can visualize which terms are more common than others, and change their initial choices. Figure 6 shows the thesaurus-based tag cloud creation process, as implemented in CodeGenie (see Section 4).

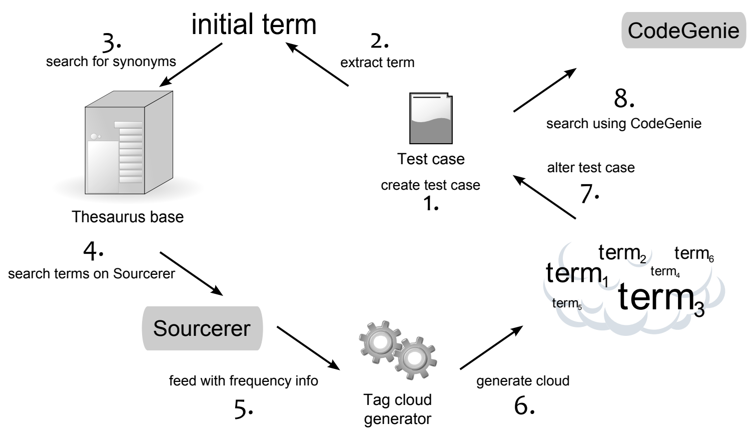

Figure 6: Thesaurus-based tag cloud creation process.

First, according to the TDCS approach, the user creates a test set for the desired functionality. Then, the name of the method to which the test cases are targeted is extracted to form the initial terms. In this process, de-camelizing of the method name is used to extract specific words. The synonyms of this term are looked up in the thesaurus base (current version of CodeGenie uses WordNet [Miller, 1995]). Each of the returned synonyms is searched in the code repository – in our case, in Sourcerer – and information about their frequency is given to the tag cloud generator. The tag cloud is then generated and shown to the user, who has the option of changing the method name used in the test cases with a search and replace task. Finally, the user can rerun the search using the more adequate term.

Initially, one might think that using synonyms to search for source code would introduce too much noise. However, since we make sure these terms are searched in a source code repository – more specifically in full qualified names of code entities –, the tag cloud will only present terms related to code. In this way, the thesaurus-based tag clouds allow "taking a peek" at the repository before running the actual search. Such a quick look makes the search more prone to return better results.

4 Implementation

To create thesaurus-based tag clouds, we need a synonyms database. For that reason, we have set up a module that connects with a WordNet thesaurus instance, which provides the synonyms. English was used because the majority of code available on Sourcerer is written in that language. However, it is important to note that other languages could be used, and switching between them would be straightforward. Also note that other types of dictionaries can be used, such as domain-specific thesauri.

Once the synonyms of a term are gathered, to create the tag cloud we need to calculate the frequency in which they appear in the Sourcerer's code base. To calculate such frequency, we use Sourcerer's search service, whose input is a query in the format: *contents:(term)*. The *contents* field limits the search to packages, classes, and method names of the entities in the data base. Such query returns a set of all occurrences of the given term, allowing for a straightforward calculation of its frequency. However, such approach could bring performance problems because each synonym term would have to be searched individually by the search service. Depending on the number of terms, the tag cloud generation would be inefficient because it would require too much communication time with the server. To cope with such problem, we used the *OR* operator supported by *Lucene*[4], a high-performance, full-featured text search engine library written in Java, which is used by Sourcerer. Such operator supports the inclusion of several terms in the *query*, resulting in a set containing all occurrences of all searched terms. The query format with the use of the *OR* operator is the following: *contents:(*$term_1$ *OR* $term_2$ *OR* $term_3$*)*.

Another characteristic that can be extracted from the results is the the position of the term in the entity's full qualified name. It is clear that the more to the right the term appears, the more the probability of the entity to be related to the searched functionality. For instance, suppose the synonym terms a user is looking for are *sum* and *calculation*, and the returned entities containing such terms are *br.com.calculationapp.utils.QuickSort.sort* and *br.com.sumapp.utils.Math.sum*. It is clear that the first term is more likely to define the searched functionality, since it matches a method name in the repository. Therefore, such term should be enhanced in the cloud. To represent such characteristic, we also applied color intensity: the higher the weight of the term

[4] `http://lucene.apache.org/core/` - accessed in 06-27-2012.

in the result set – that is, the more frequent they appear in the repository –, the higher the intensity of green in its presentation in the cloud.

The tag cloud interface was created using the Zest/Cloudio[5] Java library, which supports the creation of clouds in different sizes and colors. Zest/Cloudio also supports dimensioning of the cloud and selection of terms.

5 Evaluation

To evaluate the approach presented in this paper, we have conducted an applicability study and a controlled experiment. The first study was conducted as a preliminary evaluation, to check whether the tag clouds could simply help improve the results sets for a sample of auxiliary functions. Then, to further investigate the performance of the proposed approach, we have conducted a controlled experiment involving a relatively large sample of human subjects and auxiliary functions. Next we describe our studies.

5.1 Applicability study

In order to check whether using the thesaurus-based tag clouds would improve the result set, we have developed test cases to search for six different functions, according to the TDCS approach. The selected functions are auxiliary features commonly used in software projects. Two recent studies have shown that auxiliary functions – also called *programming tasks* – are frequently targeted by queries issued to a popular and large code search engine [Bajracharya and Lopes, 2012]; and that they are important for software projects in general [Lemos et al., 2012]. A senior computer science undergraduate student with good Java knowledge developed the sample test sets. Since he is a non-native English speaker, his choices of keywords might not be the most adequate.

Based on the test sets, we searched for the desired functions using CodeGenie. The number of returned results was recorded. Then, to compare the initial outcome to the results using our approach, we generated tag clouds based on the keywords extracted from the same test sets, and replaced the method names with the most relevant term in the clouds. We then executed the search again in CodeGenie, and recorded the results.

The searches that were made using CodeGenie considered the class and method names of the desired functionality to be reused. The selected class and method names were the following: *Calculation.total*, a function to add two integers; *Array.arrange*, a function to sort elements in an array; *Array.separate*, a function to split integer elements of an array; *File.weave*, a function to merge

[5] `http://wiki.eclipse.org/Zest/Cloudio` - accessed in 06-27-2012.

contents of two files; *QueryUtil.same*, a function to check query strings equality; and *Util.invert*, a function to reverse strings.

Figure 7 shows the generated tag clouds. Based on these clouds, the initially selected terms were replaced in the following manner: *total* was replaced by *sum* (see Figure 7(a)); *weave* was replaced by *merge* (see Figure 7(b)); *arrange* was replaced by *sort* (see Figure 7(c)); *separate* was replaced by *split* (see Figure 7(d)); and *same* was replaced by *equal* (see Figure 7(e)).

(a) *total* (b) *weave*

(c) *arrange* (d) *separate*

(e) *same* (f) *invert*

Figure 7: Tag clouds for the sample terms.

Table 1 the number of returned results for each case. Note that in all six samples the use of tag clouds enlarged significantly the number of returned results (almost tenfold, on average). A paired Wilcoxon signed-rank test revealed that the means are significantly different at 95% confidence level (p-value=0.01563). We applied the Wilcoxon test because a Shapiro-Wilk normality test on our

Table 1: Results of the exploratory evaluation.

Function	**without TC**	**with TC**	**Difference**
Calculation.total	26	68	42
Array.arrange	1	180	179
Array.separate	6	24	18
File.weave	2	80	78
QueryUtils.same	2	6	4
Util.invert	0	2	2
Avg.	**6.17**	**60**	**53.84**

Legend: TC = Tag Cloud.

data showed evidence that the results with the use of tag clouds do not follow a normal distribution (p-value=0.002674). The usefulness of our approach is particularly evident in the sixth example, where no results were returned before using the tag cloud, and two adequate results were returned afterwards.

By analyzing the results, we noted that our approach increases *recall*, but not necessarily *precision*. However, we believe the TDCS process itself supports high precision, when we consider the use of test cases and the interface of the desired function in the queries (a previous evaluation shows evidence of this [Lemos et al., 2011]: all candidates that passed the tests in the study were relevant). These two mechanisms filter out unwanted results: only candidates that match the desired interface and are successful against tests can be considered to implement the intended function. For instance, in the second example where 180 results were returned after the use of tag clouds, by executing the test cases we can exclude spurious results, and a single working candidate is enough for the success of the search. This does not mean we need to test all candidates, because experimenting with some will probably be sufficient to reach a working result (remember that for the actual search, besides running the tests, we also consider parameter and return types in the query, so it is unlikely to have the majority of returned results as irrelevant).

We noticed the thesaurus-based tag clouds can be specially useful for non-native speakers. For instance, consider the examples shown in Table 1. As commented earlier, an undergraduate student with good Java knowledge chose the terms used in the searches. He is a Brazilian Portuguese native speaker. It makes sense for him to choose the word "separate" to define the splitting of contents of an array because the most adequate word in Portuguese for this function would be *separar*. Also, *invert* makes much more sense than *revert*, because the Portuguese word *inverter* is the one used with the intended meaning, while *reverter* is more frequently used to mean *revert* in the sense of going back to a previous state. As Portuguese is a Latin-based language, the closest words in English more natural to be chosen are the likewise Latin-based "separate" and "invert". However, the tag cloud helps choosing a more common English word used to define such functions, *i.e.*, "split" and "reverse".

In any case, we believe the thesaurus-based tag clouds can be useful not only for non-native speakers, but for anyone using CodeGenie. It is clear that developers are not always aware of the most common term used to define a given function, even when the repository is written in a native language. This is specially the case when developers are unfamiliar with the domain they are currently working on.

5.2 Controlled experiment

We have conducted an experiment to look for evidence about the performance of the proposed approach[6]. At this time we wanted to include a larger number of human subjects and functions, and also an unbiased repository. The research question we have defined for our study was the following: **RQ**: *In the context of Test-Driven Code Search, can our thesaurus-based tag cloud approach improve searches in terms of recall and precision, compared to when it is not applied?* Our investigation develops in terms of two hypotheses derived from this research question. The null (0) and alternative (A) definitions of each hypothesis are described in Table 2.

Table 2: Hypotheses formulated for our experiment.

	Null hypothesis (0)	**Alternative Hypothesis (A)**
H_1	$\text{Recall}_{\text{wTC}} = \text{Recall}_{\text{woTC}}$	$\text{Recall}_{\text{wTC}} > \text{Recall}_{\text{woTC}}$
H_2	$\text{Precision}_{\text{wTC}} = \text{Precision}_{\text{woTC}}$	$\text{Precision}_{\text{wTC}} > \text{Precision}_{\text{woTC}}$

Legend: H = Hypothesis, wTC = with our Tag Cloud approach, woTC = without our Tag Cloud approach.

Repository. To have an unbiased repository, we decided to index 100 Java projects used by Fraser and Arcuri [Fraser and Arcuri, 2012] to evaluate test data generation techniques. The projects were randomly selected from SourceForge in a benchmark statistically sound and representative for open source projects (called SF100[7]).

With respect to the set of functions selected for our experiment, we looked into features that were used in previous code search studies [Hoffmann et al.,

[6] The same experimental setup (repository, survey data, sample functions, etc.) was used before to evaluate an automatic query expansion approach [Lemos et al., 2014]. However, the goals and procedure here are different, since the study targets a different approach.

[7] `http://www.evosuite.org/sf100/` - 08/08/13

2007; Reiss, 2009; Lemos et al., 2011]. To narrow down the set to a collection of realistic functions, we selected only those that appeared at least twice in our target repository. In this way we could have more evidence that these functions are in fact implemented and used in real projects. The functions were manually searched in our repository, by using several keywords and applying text-based matching, and careful inspection of the hits. This procedure was important to be able to measure recall and precision in our study. The functions selected for our study together with the frequency in which they appear in the target repository are listed in alphabetical order in Table 3.

Table 3: Functions selected for our study.

#	**Description**	**Freq.**
1	Computing the MD5 hash of a string	4
2	Decoding a URL	8
3	Encoding Java strings for HTML displaying	13
4	Encrypting a password	5
5	Filtering folder contents with specific file types	28
6	Generating the reverse complementary DNA seq.	5
7	Joining a list of strings in a single string	4
8	Revert a text string	2
9	Rotating an image	3
10	Saving an image in JPG format	3
11	Scaling an image	8
12	Zipping files	5

Queries. To adequately evaluate our approach, we had to look into queries that users would really use while performing test-driven code search; that is, we could not guess which interfaces they would pick in their test cases without biasing the study. Therefore, we conducted a survey to collect realistic interface definitions that users would use in their queries. Users were presented with the functions' descriptions in their native language, and were required to define what interfaces they would use to search for those functions. We asked them to guess the return type, name, and parameter types of the method that would implement that function, the exact information that we need while to do searches while performing TDCS. Then, to measure precision and recall for those queries, we conducted the searches by using the interfaces defined by the users.

Our survey included questions about the English knowledge level of the participant, whether they were students or professional developers, and in what field they worked in. 36 subjects responded to our survey, from them, 24 were senior Computer Science undergraduate students and 12 were professional developers; 14 had advanced English knowledge and 22 had basic English knowledge (we considered that the participants had advanced English knowledge when they could write good English, as stated by themselves). Since each subject guessed interfaces for each of the 12 target functions of our experiment, we collected a total of 432 queries.

Experimental design and procedure. For the conducted experiment, we adopted the *repeated measures* experimental design, where each query was evaluated before and after applying the tag clouds approach. Such type of design supports more control to the variability among the subjects (in this case, the queries) [Montgomery, 2006].

In our study, each query was used to search for a function with and without the application of the tag cloud approach. In this case, *paired* statistical hypothesis tests are more adequate, because they can compare measures within items rather than across them. Paired tests are considered to greatly improve precision when compared to their unpaired counterparts [Montgomery, 2006]. Before choosing the more adequate test to be applied, we must verify the normality of our observations. A Shapiro-Wilk test indicated that the observations in our experiments did not follow a normal distribution. Therefore, we decided to apply the Wilcoxon/Mann-Whitney non-parametric signed-rank paired test, which does not assume normal distributions [Shull et al., 2007].

For our statistical tests, we have adopted the most common used significance level of 0.05. Thus, our analyses consider p-values below 0.05 significant. For all statistical tests we used the R language and environment[8].

Metrics. To evaluate the performance of the queries while applying the proposed tag clouds approach, we adopted the most commonly used information retrieval metrics of *recall* and *precision*. In our context, recall is defined as the intersection between the number of relevant functions (that is, functions in the repository that implement the desired feature) and the number of retrieved functions, over the number of relevant functions; and precision is defined as the intersection between the number of relevant functions and the number of retrieved functions, over the number of retrieved functions. Recall is a measure of completeness or quantity, whereas precision is a measure of exactness or quality.

Recall and precision, as used in our context, are formally defined as follows:

$$\textbf{recall} = \frac{|\text{relevant functions} \cap \text{retrieved functions}|}{|\text{relevant functions}|}$$

$$\textbf{precision} = \frac{|\text{relevant functions} \cap \text{retrieved functions}|}{|\text{retrieved functions}|}$$

[8] `http://www.r-project.org/` - 08/05/13

We have also set an upper bound to the size of the result set while performing the searches to 100. We believe this is a reasonable boundary since it is unrealistic to think that a user would go over more than such amount of code candidates while looking for a function. Moreover, we also believe this is an adequate number of results that could be automatically tested by using TDCS. A result set larger than this would certainly significantly reduce performance.

Procedure. To simulate the use of the tag clouds approach, we have first run the queries defined by the users without the approach, and afterwards with the approach. In the second run, we have followed the following procedure to simulate a realistic scenario of the use of the tag clouds: we ran the original query and, if no good results were returned, we generated the synonyms for each of the terms in the method name part of the query, and searched for each of these terms in the repository. We then selected the top 3 synonyms that appeared most in the repository (these would be the terms that would appear larger in the tag cloud). Then, to simulate the user selection of the terms in the cloud, we generated combinations of the top 3 synonyms and the original terms used in the query, and searched again in the repository, by replacing the original method name with the combination. If for a given combination any good result was returned, we calculated the recall and precision for the new query, and recorded it in our results table. This procedure mimics the scenario where the user would try to find something by using TDCS and, if no good results was returned, would generate the tag cloud and replace original terms by more adequate ones in the query.

Results and analysis. Table 4 presents the main results of our experiment. We can clearly see that the tag cloud approach helped improve both recall and precision: recall was improved by almost 29% and precision was improved by almost 14%[9]. To check whether the observed differences were statistically significant, we ran the Wilcoxon/Mann Whitney statistical test for each metric. For recall, the test indicated a significant difference at 95% confidence level (df = 431, p-value = 0.0000587). Such result favors the alternative hypothesis H_1-A that recall is improved when the tag cloud approach is applied. For precision, the test also indicated a significant difference at the same confidence level (df = 431, p-value = 0.0000587). Such outcome favors the equally alternative hypothesis

[9] Although the gain was significant, in particular for recall, outcomes can be considered low (4-6% of recall and 0.8-0.9% of precision). This can be explained in part by our relatively small repository (in comparison with large-scale repositories such as ohloh, which indexes around 20,028 active projects [Nagappan et al., pear]). Moreover, the interface-driven queries are very specific: users want functions with a determined interface, so it is generally hard to reach relevant implementations. Another aspect that must be taken into account is that even when queries are successful in matching suitable functions, it is generally impossible to reach 100% of the relevant functions, because many times the occurrences in the repository possess different interfaces, which are mutually exclusive. For instance, if a user defines a query to find a function that returns an *int*, a relevant implementation in the repository that returns *void* will never be matched.

H_2-A that precision is improved when the tag cloud approach is applied.

Table 4: Main results of our experiment.

	Recall		**Precision**	
	without TC	**with TC**	**without TC**	**with TC**
Mean	0.04526749	0.06355453	0.008132324	0.009429222
Difference	28.77%		13.75%	

Legend: TC = Tag Clouds.

Table 5 presents the method names used in the queries for which the tag cloud approach helped improving the searches, before and after the application of the approach. Several subjects chose the inadequate term "*invert*" for the string reversion function (19 in total). Another interesting case was the use of the term "*unite*" for the string merging function: in this case the tag cloud helped selecting the more adequate term *merge*. For the other instances, the tag cloud helped indicating the simple forms of the terms: instead of *files*, the singular *file*, and instead of the inflected *encrypted*, the infinitive *encrypt*.

Table 5: Method names that were improved by the tag cloud approach.

Original term	**Suggested term**
unite	merge
invert	reverse
filterFiles	filterFile
encrypted	encrypt

Threats to validity. A threat related to internal validity that may have affected our experiment was the lack of control of the subjects' programming skills. However, we believe the repeated measures design decreases the impact of such threat (*i.e.*, the subjects' outcomes are compared within themselves before and after applying the tag-cloud approach).

A characteristic of our experiment that might have impacted its external validity is the use of students as subjects. In fact, some studies have shown opposite trends for students and professionals (*e.g.*, Arisholm and Sjoberg [Arisholm and Sjøberg, 2003]). However, according to other authors, students play an important role in experimentation in the field of software engineering [Basili et al., 1999; Kitchenham et al., 2002]. On the other hand, 12 professional developers also took part in our experiment (1/3 of our sample). We believe the inclusion of such group in our study might have circumvented this threat.

Our experiment included only Brazilian subjects, which might also have affected its external validity; *i.e.*, our results might not be generalized to the population of non-native speakers in general. However, even when we consider only the population of Portuguese speaking developers, we have a considerable amount of people that could benefit from our approach. For instance, in Brazil alone there are approximately 400,000 software developers[10]. Moreover, it is quite probable that our results might also be generalizable to native Spanish speakers, because the two languages are very similar in their relation to English. For instance, many Spanish-speaking natives would also probably choose the word *invert* in the case of the *string reversion* function discussed before, because in that language the word *invertir* would be the most frequently used with that meaning. Since Spanish is considered the second most spoken language in the world[11], a multitude of other developers could thus benefit from our approach.

Another threat to the external validity of our experiments is the representativeness of the selected functions. One might argue that the functionalities selected do not represent the population of functions in general. We agree that our results might not scale to more complex functions. However, we believe that auxiliary functions are important to software development in general, as discussed by Lemos et al. [Lemos et al., 2012]. Also, as commented before in this section, this type of function is among the most popular category of features searched in code search engines [Bajracharya and Lopes, 2012].

6 Related Work

Software reuse has been widely explored in software engineering since the 1960s [McIlroy, 1969; Krueger, 1992; Mili et al., 1995; Reiss, 2009]. Several aspects of reusability make it a hard problem, including creating reusable assets, finding these assets, and adapting them to a new application [Reiss, 2009]. Several approaches to solve this problem have been proposed, but only recent work explore code available in open source repositories. Next, we sample some software reuse and query expansion proposals, comparing them to TDCS and the approach presented in this paper.

Code reuse. An approach similar to TDCS was proposed by Podgurski and Pierce [1993]. Behavior Sampling (BS) is a retrieval technique that executes code candidates on a sample of inputs, and compares outputs to an oracle provided by the searcher. However, differently from TDCS, inputs for the desired functions are randomly generated and expected outputs have to be supplied by users.

[10] Information available at `http://tinyurl.com/ppuaq22`, accessed on 09/10/2013.
[11] Information available at `http://tinyurl.com/yrmnyw`, accessed on 09/11/2013.

TDCS implements the the ability to retrieve code based on arbitrary tests, an extension to BS considered by Podgurski & Pierce. PARSEWeb [Thummalapenta and Xie, 2007] is a tool that combines static analysis, text-based searching, and input-output type checking for a more effective search. Moreover, the specific tag clouds approach presented in this paper could also support BS, when querying the code candidates to be run against the input sample.

Reiss [2009] argues that most early reuse techniques did not succeed because they required either too little or too much specification of the desired feature. Signature or type matching used by themselves do not seem to be effective, although PARSEWeb shows that it can provide more interesting results in combination with textual search. Full semantic matching requires the specification of too much information, and is thus difficult to accomplish. TDCS uses test cases, which are generally easy to check and provide.

In a recent work, Reiss [2009] also incorporates the use of test cases and other types of low-level semantics specifications to source code search. He also implements various transformations to make available code work in the current user's context. However, in his approach, there is no slicing facility and therefore only a single class can be retrieved and reused at a time. Moreover, the presented implementation is a web application, which also requires code results to be copied and pasted into the workspace in an *ad hoc* way. CodeGenie has the advantage of being tightly integrated with the IDE: code candidates can be seamlessly woven to and unwoven from the workspace. Moreover, Reiss' approach does not incorporate anything similar to the tag clouds approach presented in this paper to help the user to reformulate queries.

Hummel et al. [2008] also developed a tool similar to CodeGenie, named Code Conjurer. The tool is more proactive than CodeGenie, in the sense that code candidates are recommended to the user without the need of interaction. However, the slicing facility presented by CodeGenie is more fine grained – at method level –, and Code Conjurer does not deal with the vocabulary mismatch problem targeted by the tag cloud-based approach presented in this paper. In fact, the tool implements an automated adaptation engine, but it deals only with the interface specification (*i.e.*, not with keywords).

Modern CASE tools are bringing sophisticated search capabilities into the IDE, extending traditionally limited browsing and searching capabilities [Holmes and Murphy, 2005; Mandelin et al., 2005; Sindhgatta, 2006; Poshyvanyk et al., 2006; Sahavechaphan and Claypool, 2006; da Silva Jr. et al., 2012]. These tools vary in terms of the provided features, but some common ideas that are prevalent among them are the use of the developer's current context to generate queries and the integration of ranking techniques for the search results. CodeGenie also shares such features, bringing the use of test cases to provide a more sophisticated solution.

Tag clouds are also used in the Sourcerer API Search (SAS) [Bajracharya et al., 2010a]. The difference between SAS and CodeGenie is that the latter is tightly integrated with the IDE, and thus the user can access tag clouds directly from the development environment. Moreover, CodeGenie makes use of test cases to form queries, while SAS is mostly based on keywords. Another difference is that tag clouds generated by SAS are not based on thesauri, but on an API similarity algorithm.

Query expansion. The application of Natural-Language Processing (NLP) to code and concept search has been proposed by earlier approaches. For instance, Shepherd et al. [2007] combines NLP and structural program analysis to locate and understand concerns in a software system. The basic difference from our approach is that the concern location is targeted towards code in a local project, not in a code repository. Moreover, Shepherd et al.'s approach [Shepherd et al., 2007] requires more interaction, since similar terms have to be chosen by the user iteratively to construct the expanded queries. CodeGenie supported by tag clouds only require the user to choose a more adequate term once to reissue the search.

Gay et al. [2009] have proposed an IR-based concept location approach that incorporates relevance feedback from the user to reformulate queries. The proposed approach, however, is not directed to code reuse, but for concept location within a given project. Moreover, such as Shepherd et al.'s approach, is requires more user interaction.

More recently, Yang and Tan [2012] proposed an approach that identifies word relations in code, by leveraging the context of words in comments and code. The idea is to find words that are related in code, but not in English. For such pairs, lexicals like Wordnet cannot help. Their approach could also be applied to CodeGenie, to improve the dictionary used in the formation of tag clouds, with a secondary code-related synonyms database. In fact, we are currently working on an extension to our approach that incorporates such idea.

Most of the query reformulation approaches that were proposed in the past focus on concept location within a project, and mainly to identify code that must be dealt with in a particular software engineering task. The difference between such approaches and ours is that our goal is to help finding code – more specifically, methods – inside large open source repositories, with the intent of reusing it.

However, other sophisticated NLP techniques incorporated by them could also be explored in the generation of our tag clouds. For instance, *morphology changes* could be applied to search other morphological forms of a given term. This could improve the effectiveness of our thesaurus-based tag clouds.

7 Conclusion and Future Work

In this paper we have presented an approach that applies thesaurus-based tag clouds to improve TDCS. The tag clouds are formed by using an initial term extracted from the input test set, and generating synonyms from this term. Synonyms are then looked up in a code base, being presented in the tag cloud according to their frequency and weight in the repository. Initial evaluation has shown that the tag clouds can enlarge the set of returned candidates of a given search whose initial term would not be the most adequate. A controlled experiment with 36 subjects – including professional developers – also presents evidence that recall and precision can be improved by the approach: recall by $\sim$28%, on average; and precision by $\sim$14%, on average.

Future work includes generating tag clouds based on API similarity instead of synonyms. More support for non-native speakers to better explore TDCS is also an interesting line of investigation. Dictionaries could be used to translate the initial term to the equivalent word in the intended language. This type of support would allow users to improve their chances of finding relevant code.

Another line of future work we are currently investigating is automatically expanding the code search queries. The idea shares the basic principle of the thesaurus-based tag clouds, but instead of generating the cloud, the query itself is automatically expanded with similar terms. Such approach would skip the tag cloud examination and changing of initially developed test cases. However, the tag cloud approach would still be relevant in cases where the developer wants to check for terms in the repository before creating the test cases for the search.

Acknowledgements

The authors would like to thank FAPESP for financial support (Otavio Lemos, grant 2010/15540-2).

References

[Arisholm and Sjøberg, 2003] Arisholm, E. and Sjøberg, D. I. K. (2003). A controlled experiment with professionals to evaluate the effect of a delegated versus centralized control style on the maintainability of object-oriented software. Technical Report 6, Simula Research Laboratory.

[Bajracharya et al., 2010a] Bajracharya, S., Ossher, J., and Lopes, C. (2010a). Searching api usage examples in code repositories with sourcerer api search. In *Proc. of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, SUITE '10, pages 5–8, New York, NY, USA. ACM.

[Bajracharya and Lopes, 2012] Bajracharya, S. K. and Lopes, C. V. (2012). Analyzing and mining a code search engine usage log. *Empirical Softw. Engg.*, 17(4-5):424–466.

[Bajracharya et al., 2010b] Bajracharya, S. K., Ossher, J., and Lopes, C. V. (2010b). Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proc. of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 157–166.

[Basili et al., 1999] Basili, V. R., Shull, F., and Lanubile, F. (1999). Building knowledge through families of experiments. *IEEE Trans. Softw. Eng.*, 25:456–473.

[Beck, 2002] Beck, K. (2002). *Test driven development: By example.* Addison-Wesley Professional.

[Cremonini, 2002] Cremonini, L. (2002). Web 2.0 map. Available at: `http://www.railsonwave.com/2007/1/2/web-2-0-map` (accessed 29/03/2011).

[da Silva Jr. et al., 2012] da Silva Jr., L. L. N., de Oliveira Alexandre Plastino, T. N., and Murta, L. G. P. (2012). Vertical code completion: Going beyond the current ctrl+space. In *Proc. of the 6th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pages 81–90. IEEE Computer Society.

[Erdogmus et al., 2005] Erdogmus, H., Morisio, M., and Torchiano, M. (2005). On the effectiveness of the test-first approach to programming. *IEEE Trans. Softw. Eng.*, 31(3):226–237.

[Fischer et al., 1991] Fischer, G., Henninger, S., and Redmiles, D. (1991). Cognitive tools for locating and comprehending software objects for reuse. In *Proc. of the 13th ICSE*, ICSE '91, pages 318–328, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Fraser and Arcuri, 2012] Fraser, G. and Arcuri, A. (2012). Sound empirical evidence in software testing. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 178–188, Piscataway, NJ, USA. IEEE Press.

[Furnas et al., 1987] Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T. (1987). The vocabulary problem in human-system communication. *Commun. ACM*, 30:964–971.

[Gay et al., 2009] Gay, G., Haiduc, S., Marcus, A., and Menzies, T. (2009). On the use of relevance feedback in ir-based concept location. In *Proc. of the IEEE International Conference on Software Maintenance*, pages 351–360. IEEE.

[Hoffmann et al., 2007] Hoffmann, R., Fogarty, J., and Weld, D. S. (2007). Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, UIST '07, pages 13–22, New York, NY, USA. ACM.

[Holmes and Murphy, 2005] Holmes, R. and Murphy, G. C. (2005). Using structural context to recommend source code examples. In *ICSE '05: Proc. of the 27th international conference on Software engineering*, pages 117–125, New York, NY, USA. ACM Press.

[Hummel et al., 2008] Hummel, O., Janjic, W., and Atkinson, C. (2008). Code conjurer: Pulling reusable software out of thin air. *IEEE Softw.*, 25(5):45–52.

[Kitchenham et al., 2002] Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., Emam, K. E., and Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28:721–734.

[Krueger, 1992] Krueger, C. W. (1992). Software reuse. *ACM Comput. Surv.*, 24(2):131–183.

[Lemos et al., 2014] Lemos, O., de Paula, A., Zanichelli, F., and Lopes, C. (2014). Thesaurus-based automatic query expansion for interface-driven code search. (submitted for publication).

[Lemos et al., 2011] Lemos, O. A. L., Bajracharya, S., Ossher, J., Masiero, P. C., and Lopes, C. (2011). A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Inf. Softw. Technol.*, 53:294–306.

[Lemos et al., 2012] Lemos, O. A. L., Ferrari, F. C., Silveira, F. F., and Garcia, A. (2012). Development of auxiliary functions: should you be agile? an empirical assessment of pair programming and test-first programming. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 529–539, Piscataway, NJ, USA. IEEE Press.

[Linstead et al., 2009] Linstead, E., Bajracharya, S., Ngo, T., Rigor, P., Lopes, C., and Baldi, P. (2009). Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowl. Discovery*, 18:300–336.

[Mandelin et al., 2005] Mandelin, D., Xu, L., Bodík, R., and Kimelman, D. (2005). Jungloid mining: helping to navigate the api jungle. In *PLDI '05: Proc. of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 48–61, New York, NY, USA. ACM Press.

[McIlroy, 1969] McIlroy, M. D. (1969). Mass produced software components. In Naur, P. and Randell, B., editors, *Proc. of NATO Softw. Eng. Conference*, pages 138–150. Garmisch, Germany.

[Mili et al., 1995] Mili, H., Mili, F., and Mili, A. (1995). Reusing software: Issues and research directions. *IEEE Trans. Softw. Eng.*, 21(6):528–562.

[Miller, 1995] Miller, G. A. (1995). Wordnet: a lexical database for english. *Commun. ACM*, 38(11):39–41.

[Montgomery, 2006] Montgomery, D. C. (2006). *Design and Analysis of Experiments*. John Wiley & Sons.

[Nagappan et al., pear] Nagappan, M., Zimmermann, T., and Bird, C. (2013 (to appear)). Diversity in software engineering research. In *Proc. of the 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2013)*, ESEC/FSE '13, New York, NY, USA. ACM.

[Podgurski and Pierce, 1993] Podgurski, A. and Pierce, L. (1993). Retrieving reusable software by sampling behavior. *ACM Trans. Softw. Eng. Methodol.*, 2(3):286–303.

[Poshyvanyk et al., 2006] Poshyvanyk, D., Marcus, A., and Dong, Y. (2006). JIRiSS - an eclipse plug-in for source code exploration. In *ICPC '06: Proc. of the 14th IEEE International Conference on Program Comprehension*, pages 252–255, Washington, DC, USA. IEEE Computer Society.

[Reiss, 2009] Reiss, S. P. (2009). Semantics-based code search. In *ICSE '09: Proc. of the 2009 IEEE 31st International Conference on Software Engineering*, pages 243–253, Washington, DC, USA. IEEE Computer Society.

[Rivadeneira et al., 2007] Rivadeneira, A. W., Gruen, D. M., Muller, M. J., and Millen, D. R. (2007). Getting our head in the clouds: toward evaluation studies of tagclouds. In *Proc. of the SIGCHI conference on Human factors in computing systems*, pages 995–998, New York, NY, USA. ACM.

[Sahavechaphan and Claypool, 2006] Sahavechaphan, N. and Claypool, K. (2006). Xsnippet: mining for sample code. In *OOPSLA '06: Proc. of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 413–430, New York, NY, USA. ACM Press.

[Shepherd et al., 2007] Shepherd, D., Fry, Z. P., Hill, E., Pollock, L., and Vijay-Shanker, K. (2007). Using natural language program analysis to locate and understand action-oriented concerns. In *Proc. of the 6th international conference on Aspect-oriented software development*, AOSD '07, pages 212–224, New York, NY, USA. ACM.

[Shull et al., 2007] Shull, F., Singer, J., and Sjøberg, D. I. (2007). *Guide to Advanced Empirical Software Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Sinclair and Cardew-Hall, 2008] Sinclair, J. and Cardew-Hall, M. (2008). The folksonomy tag cloud: When is it useful? *J. Inf. Sci.*, 34(1):15–29.

[Sindhgatta, 2006] Sindhgatta, R. (2006). Using an information retrieval system to retrieve source code samples. In Osterweil, L. J., Rombach, H. D., and Soffa, M. L., editors, *ICSE*, pages 905–908. ACM.

[Thummalapenta and Xie, 2007] Thummalapenta, S. and Xie, T. (2007). Parseweb: a programmer assistant for reusing open source code on the web. In *ASE '07: Proc. of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213, New York, NY, USA. ACM.

[Yang and Tan, 2012] Yang, J. and Tan, L. (2012). Inferring semantically related words from software context. In *Proc. of the 9th IEEE Working Conference on Mining Software Repositories*, pages 161–170. IEEE.